Laboratory Learning Material

on

`Verilog` HDL

IC Design Group

Central Electronics Engineering Research Institute

Pilani – 333 031 (Rajasthan)

November, 2004

Sponsored by Department of Information Technology

Ministry of Communications and Information Technology

(Government of India)

Under the Project

"Special Manpower Development for VLSI Design and Related Software"

Contents

List of Figures

Chapter 1

About Laboratory Learning Material on Verilog HDL

1.1 Code of the LM

L3.

1.2 Participating Faculty

1. Dr. Chandra Shekhar, CEERI-Pilani.
2. Mr. Raj Singh, CEERI-Pilani.
3. Mr. Sudhir Kumar, CEERI-Pilani.

1.3 Reviewer's Details

```
Prof. S. Srinivasan
Department of Electrical Engineering
IIT-Madras
Chennai -- 600 036
```

1.4 Objective

This lab is designed to familiarize the student with the constructs, modeling styles and coding aspects related to simulation and synthesis for `Verilog` HDL.

It is meant to give the students a good understanding of the issues and steps involved in using `Verilog` by way of solved examples and suggested exercises.

The material developed can also serve as a basis and model for designing student projects and the associated methodology of carrying out the same.

1.5 Pre-requisite LM/Courses/Background

1. Exposure to `Verilog` HDL.
2. Basic course on "Digital Logic Design."

1.6 Coverage

These contents are intended to be covered in 10-12 lab turns (of 3 hours each).

1. Experiments related to language semantics :
    (a) Time Control : delay operator, event control.
    (b) Assignment Types : procedural, blocking, nonblocking, continuous.
    (c) Delay through combinational logic and nets.
2. Behavioural Modeling (examples and suggested exercises).
3. Data-flow Modeling (examples and suggested exercises).
4. Structural Modeling (examples and suggested exercises).
5. RT-Level Modeling (examples and suggested exercises).
6. Mixed-Style Modeling (example).
7. Modeling of state machines and sequential logic (examples and suggested exercises).
8. Coding of test benches.
9. Coding style for synthesis.

10. Entering design constraints for synthesis; Generating timing reports; CLB/gate usage reports; Identifying suitable FPGA device (Xilinix) for design implementation. ??

11. Suggestions for mini-project topics.

1.7 Recommended CAD Tools and Platforms

1. MTI's ModelSim for simulation.

2. Xilinx Foundation tool set for synthesis.

3. Advance projects and Masters level theses students can use Cadence and/or Synopsys tools. For implementation on Xilinx devices, use Xilinx Alliance tool set with Cadence/Synopsys/ModelSim.

Chapter 2

Verilog Tutorial

2.1 Verilog Features

- It provides a set of constructs to create concurrent processes to model the inherent concurrency in digital hardware.
- It supports hierarchical design – for clarity and brevity in the description of complex designs.
- It supports modeling of behavioural, structural and mixed views of designs over a range of design abstraction levels : Algorithmic, Register Transfer (RTL), Boolean Expression, State-Table/ Truth-Table levels for behavioural description; hierarchical block level, gate level and switch level net lists for structural description – with a freedom of mixing behavioural and structural descriptions at different design abstraction levels in describing a single design.
- It provides explicit event control as well as delay control over the time of execution of statements in the procedural code that models the behaviour of hardware at the algorithmic level. Also, language constructs are available to support the structuring of procedural code along the familiar concepts of structured programming.
- It supports various design methodologies – Top-down, Bottom-up and Mixed.
- It supports import of additional/more accurate information on the design generated at lower levels of abstraction (not supported by the language) into design descriptions at higher levels of abstraction (supported by the language) *e.g.* back-annotation of extracted delay from layout into gate level description.
- It provides a technology-independent design environment (and additionally switch level support for MOS VLSI circuits) with a provision for import of technology-specific information through back-annotation.
- It is an IEEE standard language (1364) – open to all – encourages the development of tools around the language.

- It is human readable as well as machine readable – with precise simulation semantics.
- A design description written in the language is also an executable model of the hardware. One can, thus, create an *executable documentation* of the design.

2.2 A Brief History of `Verilog` HDL

The Verilog HDL was first introduced in 1984 as a proprietary language from Gateway Design Automation founded by Dr. Prabhu Goel. In 1989, Cadence acquired Gateway Design Automation. Subsequently, Verilog standardization was handed over to OVI who then moved to put it in public-domain by submitting the Verilog to IEEE. In 1995, IEEE 1364-1995 became the official Verilog standard. The language was revised in 2000-2001 to become IEEE 1364-2001.

2.3 Overview

In Verilog, a module defines the model of a digital hardware (whose complexity can vary from that of a complete system to a simple gate). Verilog also allows switch level models of transistors, inverters and buffers. ??

It is to be noted that Verilog uses a standard four-valued logic (0, 1, x, z) for all digital modeling. However, as we will see later, each of these logic values can have an associated drive strength (from a set of language-prescribed strengths) to reflect the strength of the driver driving the logic value.

Each module has a module name that follows the language keyword `module`. The module name is followed by a parenthesized, comma-separated list of port names (through which the module communicates with the outside world) terminated by a semi-colon. Key words, `module` and `endmodule` act as syntactic brackets that hold the module definition.

Following the module name and module port specification statement, there are declaration statements that declare the type of each port — whether `input`, `output` or `inout`.

Following the port type declaration statements there are declaration statements for registers and nets. Registers are used for assignment of values by sequential statements inside the `initial` and `always` blocks which are used to describe the behaviour of hardware algorithmically. Thus, in Verilog registers play the same role as variables do in sequential programming languages such as C.

The nets (and wire is one type of net) are used to connect the instances of constituent gates or user defined modules in the structural description of a higher level module. They are also used for assigning values of Boolean expressions by continuous assignment statements that Verilog provides for data-flow modeling of hardware.

The language provides two distinct classes of data items namely register and net — for separate uses. Registers can only be assigned values (and nets cannot be assigned values) in the block of sequential statements associated with an initial or always statement. Therefore, registers in Verilog play the same role as variables in sequential programming languages such as C.

Nets (and nets are of many different types as we will see later) can only be assigned values (and registers cannot be assigned values) by continuous assignment statements. Besides this, nets are also used to connect the ports of different gate instances and module instances, and they propagate the values driven on them by the output ports of gates and modules they connect to, to the input ports of gates and modules they are connected to.

All module ports are nets of type `wire`. A behavioural model having only `initial` and `always` statements whose associated blocks of sequential statements can assign values only to registers and cannot assign values to output ports (because ports are wires) drive values onto output ports of modules via declaring a register of the same name as the name of the output port of the module, and assigning values to this register which implicitly drives the port.

Following the declarations of registers and nets there is a set of concurrent statements that describe the module behaviourally, structurally, in a data-flow manner or in a manner that is a mix of all the three manners.

2.4 Modules

In Verilog, a `module` models a piece of digital hardware. It is also used for writing test benches.

A `module` has a name, a port list, a description of the type of each port and either a behaviour description, or a structural description or a mixed behavioural-structural description of the hardware.

```
module dEdgeFF(q, clock, data);

output q;
reg q;
input clock, data;
wire clock, data;

initial
  q = 0;
```

```
always
  @(negedge clock) #10 q = data;

endmodule

module ffNand;

wire q, qBar;
reg preset, clear;

nand #1
  g1(q, qBar, preset),
  g2(qBar, q, clear);

endmodule
```

2.5 Comments

In-line comments are given as //.

For a block comment extending over several lines, open with /* and close with */.

```
// This is a comment.

/* This shows the use of
   a block comment. */
```

2.6 Identifiers

Identifiers are names that are given to elements such as modules, registers *etc.*

An identifier is any sequence of letters, digits and the underscore symbol except that :

- The first character must not be a digit.
- The identifier must be 1024 characters or less.

Verilog is CASE SENSITIVE – upper and lower case letters are considered different.

System tasks and functions always start with the dollar($) symbol.

Verilog also permits *escaped identifiers e.g.* identifiers whose leading character is backslash(\) which is then followed by other characters (including those not permitted by Verilog in dentifiers).

The leading backslash character is not considered a part of the identifier. Escaped identifiers provide a facility in Verilog to accept names from other CAD systems with their own naming conventions.

2.7 Constants

Constant numbers can be specified in decimal, hexadecimal, octal or binary. They may optionally start with a $+$ or $-$, and can be given in one of the two forms.

Form 1 :

Unsized decimal number specified using the digits from the sequence 0 . . . 9.

Form 2 :

`ss ... s'fnn ... n`

where `ss ... s` is the bit size of the constant (specified using decimal digits)

`'f` is the base format where `f` is respectively replaced by `d` (for decimal), `h` (hexadecimal), `o` (octal) or `b` (binary). The letters may also be capitalized.

`nn ... n` is the value of the constant specified in the given base with allowable digits.

Unknown and high impedance values may be given in all but the decimal base. In each case, the `x` or `z` character represents the given number of bits of `x` or `z` *i.e.* in hexadecimal, an `x` would represent 4 unknown bits, and in octal, 3 unknown bits.

```
12'b0x0x_1101_0zx1
12'b0x0x11010zx1
792      // an unsized decimal number.
'h7d9    // an unsized hexadecimal number.
'o7746   // an unsized octal number.
12'hx    // a 12-bit unknown number.
8'hfz    // equivalent to 8'b1111zzzz.
10'd17   // a 10-bit constant with value 17.
```

2.8 Strings

A string is a sequence of characters enclosed by double quotes. It must be contained on a single line. Special characters may be specified in a string using the \ escape character as follows :

```
\n new line character.
\t tab character.
\\ is the \ character
\"␣is␣the␣" character.
```

`\ddd` is an ASCII character specified in 1 to 3 octal digits.

2.9 Integer

Integer objects are used only to perform calculations for simulation purposes – for example to turn-off the monitoring of a signal after a certain time. (Use of a register for this purpose would confuse it with the actual behaviour of the hardware). Thus, integers are not used for describing hardware behaviour. Integers are 32-bit 2's complement signed quantities.

```
// Two integers a and b.
integer a, b;

// An array of integers.
integer c[1:100];
```

2.10 Real

Real objects store floating point values in Verilog. They (like integers) are not used for modeling hardware.

2.11 Time

Time objects are 64-bit quantities physically used to store current values of simulation time.

2.12 Operators

Operators are used to perform various kinds of operations on values of data objects in an expression.

`{,}`	: Concatenation
+	: Addition
-	: Subtraction
-	: Unary Minus
`*`	: Multiplication
/	: Division
%	: Modulus (finds remainder)
>	: Greater than
>=	: Greater than or equal
<	: Less than
<=	: Less than or equal
!	: Logical negation converts a non-0 value (True) into 0; a 0 value into 1; and an ambiguous truth value into x.
&&	: Logical AND used as a logical connective. if ((a > b) && (c > d))
||	: Logical OR
==	: Logical equality
!=	: Logical inequality
===	: Case equality
!==	: Case inequality
~	: Bitwise negation
&	: Bitwise AND
|	: Bitwise OR
^	: Bitwise XOR
^~	: Bitwise XNOR
~^	: Bitwise XNOR
&	: Unary reduction AND
~&	: Unary reduction NAND
|	: Unary reduction OR
~|	: Unary reduction NOR
^	: Unary reduction XOR
~^	: Unary reduction XNOR
^~	: Unary reduction XNOR
<<	: Left shift (vacated bit positions are filled with 0s)
>>	: Right shift (vacated bit positions are filled with 0s)
?:	: Conditional (cond expr ? true-expr : false-expr)

2.12.1 Operator Precedence

There is a set of operator precedence rules, but it is best to use brackets for clarity in specifying the order in which the operators will take effect.

Operator truth tables are given in various books and the LRM. Refer to the list of books given separately as a chapter near the end of this LM.

2.13 Sequential Statements : If Statement

Example :

```
if (negdivisor)
  divisor = -divisor;

if (a < 0)
  mod_a = -a;
else
  mod_a = a;

if (ir[15:13] == 3'b000)
  pc = m[ir[12:0]];
else if (ir[15:13] == 3'b001)
  pc = pc + m[ir[12:0]];
else if (ir[15:13] == 3'b010)
  pc = pc + 1;
```

2.14 Sequential Statements : Conditional Operator Statement

Example :

```
mod_a = (a < 0) ? -a : a;
```

2.15 Sequential Statements : Case Statement

Example :

```
case (selection)
   2'b00    : $display("2'b00");
   2'b01    : $display("2'b01");
   2'b0x    : $display("2'b0x");
   2'b0z    : $display("2'b0z");
   default  : $display("undefined");
endcase
```

The case-expression is compared against the case items in their serial order literally in a bitwise manner to find a match to select the statement(s) to execute.

2.16 Sequential Statements : Casez Statement

Example :

```
casez(selection)
   2'b00    : $display("2'b00");
   2'b01    : $display("2'b01");
   2'b0x    : $display("2'b0x");
   2'b0z    : $display("2'b0z");
   default  : $display("undefined");
endcase
```

The `z` value for a bit is treated as don't care when case-expression and case items are compared for match.

What will be the print-outs generated by the `case` and `casez` for the following values of selection ? :

```
   2'b00
   2'b0x
   2'b0z
   2'bzz
   2'b??
```

2.17 Sequential Statements : Casex Statement

Both `x` and `z` are treated as don't care when case-expression is compared with case-items for a match.

2.18 Sequential Statements : Loop Statements

Verilog has 4 basic loop statements :

1. repeat.
2. for.
3. while.
4. forever.

```
repeat(count-expression) statement;

for(init-stmt; loop-end-expr; loop-update)
  statement;

while(expression) statement;

forever statement;
```

Example :

```
for(i = 16; i; i = i - 1)
  begin
    statement 1;
    . . .
    statement n;
  end
```

2.19 Exiting Loops on Exceptional Conditions

A disable statement disables or terminates any named begin – end block. Execution then begins with the statement following the block.

Example :

```
begin:LoopBlock
for(i = 0; i < n; i = i + 1)
  begin:ForBlock
  if(a == 0)
    disable ForBlock;
    // proceed with i = i + 1;
  . . . // other statements
  if(a == b)
    disable LoopBlock;  // exit for loop
  . . . // other statements
  end // ForBlock
end // LoopBlock
```

2.20 Function

This is similar to a software program's function. It may be called from within an expression and the one value it returns will be used in the expression.

Variables may be declared within the function – their scope will be the function. Functions must not include timing or event control statements. A function may also be called from within a continuous assignment.

Example :

```
function[31:0] bitwiseAND;

  input[31:0] a;
  input[31:0] b;

  begin
    bitwiseAND = a & b;
  end

endfunction
```

2.21 Task

A `Verilog` task is analogous to a software procedure. It is called from a calling statement and after execution it returns control to the next statement. Parameters may be passed to it and results returned. Local variables may be declared within it and their scope will be the task. A task is defined within a module using the `task` and `endtask` keywords. Sequential statements within the task can use all the different time control constructs.

Example :

```
module Top;

// declarations zone
reg[31:0] a, b, c;
```

```
always

  begin
    Numadd(a, b, c);
    . . . // other statements
  end

task Numadd;

  input[31:0] x, y;
  output[31:0] z;

  // if required by the algorithm,
  // local registers can also be declared.

  begin
    z = x + y;
    . . . // more statements
  end

endtask

. . . // more statements

endmodule
```

2.22 Time Control

Verilog provides 3 different types of time control constructs that can be used in a behavioural description.

- Delay construct (#)
  - To delay the execution of a statement.
  - To delay the assignment of a computed expression value to a register.
- Event Construct (@)

  To make the execution of a statement wait until the specified change(s) (event) has occurred in some other part of the circuit (outside the concurrent process containing the time control statement).
- Conditional Delay Construct :

  `wait (condition);`

To make the execution of a statement wait until the condition becomes true. If the condition is already true when the execution reaches the wait statement, execution continues beyond the wait statement and the process is not suspended.

2.23 Module Port Specifications

A port of a module can be viewed as providing a link or connection between two items, one internal to the module instance and the other external to it.

- An input or inout port cannot be declared to be of type register. Either of these ports may be read into a register using a procedural assignment statement. However, a register may only drive an inout port through a gate such as bufif0 gate.
- The output ports of a module are by definition connected to a signal source item such as a net, a register, a gate output, or a continuous assignment internal to the module.

  If an output port of a module has an internal register of the same name as the output port, the output port is implicitly connected to and driven by the register.
- The inout ports of a module are connected internally to gate outputs or inputs. Externally, only scalar or vector nets may be connected to a module's outputs.

  ```
  ahh m1(s, q, r);
  ```
- We may connect to the ports of a module instance either using an ordered list of nets or by naming the port and giving its connection.

```
wire s, q, r;
ahh mz (.b(q), .a(s), .c(r));
```

Period (.) introduces the port name defined in the module definition and the following brackets contain the net name to which it connects when the module is instantiated.

In this example, port b of instance mz of module ahh is connected to wire q.

2.24 Continuous Assignment to Nets

Continuous assignments provide a means to abstractly model combinational hardware driving value onto nets.

Continuous assignments can also have drive strength and delays associated with them – exactly like those with primitive gate instances.

General form of the assignment statement is :

```
assign drive-strength delay list-of-assignments;
```

Continuous assignments can only occur outside concurrent processes (defined by `always` or `initial` construct). A continuous assignment continuously drives a value on the net. If any input to the assign statement changes at any time, the assign statement will be re-evaluated and the value being driven on the net will change.

Example :

```
assign #10 sum = aIn ^ bIn^ cIn,
  cout = (aIn & bIn) | (bIn & cIn)
       | (aIn & cIn);

assign (strong0, strong1) #10
  sum = aIn ^ bIn ^ cIn,
  cout = (aIn & bIn) | (bIn & cIn)
       | (aIn & cIn);
```

2.25 Net and Continuous Assignment Declarations

`Verilog` permits creating data objects that cannot be assigned values by a procedural statement in a concurrent process even though their values can be read and used for computation inside the procedural code.

Data objects of this class are nets of different types which are used in structural description to connect the ports of various module instances or are driven by continuous assignments (which continuously drive a logic expression value on a net).

Nets of different types which are used to propagate values internally within a structural module are declared immediately following the port type specifications.

Nets can be scalar or vector (with a range of bits). Nets can also be declared to have net delays associated with them (this can model the delay of signal propagation over the net). Declaration of a trireg Net can also specify a charge strength for the net-which models the 'amount' of charge strength on the net. A net declaration may also have a continuous assignment placed on the net.

The different net types are :

- `wire` and `tri` : used to model connections with no logic function.
- `wand`, `wor`, `triand`, `trior` : used to model the wired logic functions.
- `tri0`, `tri1` : used to model connections with a resistive pull-up to the given supply.

- `supply0`, `supply1` : used to model the connection to a power supply.
- `trireg` : used to model the charge storage on a net.

General syntax for net declaration :

```
net-type charge-strength range delay list-of-names;
```

2.26 Assignments with Time Control

The # and @ time control discussed earlier precedes a statement. These forms of time control delay execution of the following statement until the specified simulation time.

There are two special kinds of assignment statements that have time control inside the assignment statement. These two forms are known as the blocking and nonblocking procedural assignments.

2.27 Blocking Procedural Assignment

```
var = #delay expression;
var = @(posedge onebit) expression;
var = @(negedge onebit) expression;
```

What distinguishes this from a normal instantaneous assignment is that the expression is evaluated at the current simulation time (`$time`) when the statement is reached, but the variable does not change until after the specified delay. Also (as the name implies), the execution of process is blocked (suspended) for the specified delay.

```
initial
  begin
    . . .
    a = @(posedge sysclk) b;
    . . .
  end
```

is same as

```
initial
  begin
    . . .
    temp = b;
    @(posedge sysclk) a = temp;
    . . .
  end
```

2.28 Nonblocking Procedural Assignment

The syntax for a nonblocking procedural assignment is identical to a blocking procedural assignments except that the assignment operator is <= instead of =.

```
var <= #delay expression;
var <= @(posedge onebit) expression;
var <= @(negedge onebit) expression;
```

For a nonblocking procedural assignment, the expression is evaluated at the current simulation time (`$time`) when the execution reaches the statement, but the variable is updated after the specified delay. However, execution continues with the next sequential statement at the current simulation time (`$time`).

2.29 Gate Primitives

and , **nand** , **nor** , **or** , **xor** , **xnor** , **not** ,
buf , **bufif0** , **bufif1** , **notif0** , **notif1**

Each of these gates is a generic gate – it has one output and a variable number of input determined by the number of nets/registers connected as inputs at the time of instantiation of the gate.

```
nand U1(z, a, b, c);
```

Above represents a 3-input NAND gate with output connected to `z` and inputs connected to `a`, `b`, and `c`.

```
nand U1(y, x, z);
```

Above represents a 2-input NAND gate with output connected to `y` and inputs connected to `x`, and `z`.

The first identifier in the gate instantiation is the single output or the bidirectional port and all the other identifiers are the inputs. Any number of inputs may be listed.

`buf` and `not` gates may have any number of outputs; the single input is listed last. The `buf` gate is a non-inverting buffer, and the `not` gate is an inverter.

The bufif0, buif1 notif0 and notif1 gates provide the buf and not function with a tristate enable input.

The general syntax for instantiating a gate is :

```
gate-type strength delay list-of-gate-instances;
```

```
nand (strong0, strong1) #3
  U1(z, a, b), U2(d, e, f);
```

2.30 Switch Level Transistors

Verilog provides the following switch level transistors for switch level modeling :

nmos, **pmos**, **cmos**, **rnmos**, **rpmos**, **rcmos**,
tran, **tranif0**, **tranif1**, **rtran**, **rtranif0**, **rtranif1**,
pullup, **pulldown**

2.31 Use of Parameters

Verilog permits declaring parameters (and their default values) inside modules and then writing the module description in terms of the parameters (parameterized modules).

At the time of instantiation, a parameterized module may use either the default values or may have them over-ridden by new instance specific values – through the supply of new parameter values at the time of instantiation or through a defparam block.

```
// Parameterized module definition.
module modXor(AXorB, a, b);

  parameter size = 8, delay = 15;
  output[size-1:0] AXorB;
  input[size-1:0] a, b;
  wire[size-1:0] #delay AXorB = a ^ b;

endmodule

. . .

// Use default values of size and delay.
modXor a(a1, b1, c1);
```

```
. . .

// Use new values, size = 4 and
// delay = 5 for this instance.
modXor #(4, 5) b(a2, b2, c2);

. . .

// uses new values, size = 4 and
// default value for delay.
modXor #(4) c(a3, b3, c3);
```

However, if we wanted to use the default value of the first parameter (size) and only wanted to modify the value of the second (or subsequent) parameter (delay) to be 20, we would need to write

```
modXor #(8, 20) d(a4, b4, c4);
```

2.32 defparam Statement

Using the defparam statement, all of the re-specifications of parameters can be grouped at one place within the description.

```
module Top;

// Other declarations.

modXor a(a1, b1, c1),
       b(a2, b2, c2);

endmodule

module modXor(AXorB, a, b);

  parameter size = 8, delay = 15;
  output[size-1:0] AXorB;
  input[size-1:0] a, b;
  wire[size-1:0] #delay AXorB = a ^ b;

endmodule

module annotate;

  defparam
    Top.b.size = 4,
    Top.b.delay = 5;

endmodule
```

2.33 Gate Delay Models

Gate delays can use 1, 2 or 3 values of delays.

```
// Unnamed instances are also allowed.
// rise delay = 5, fall delay = 5
not #5 (ndata, data);

// rise delay = 12, fall delay = 15
nand #(12, 15) U1(z, a, b);

// rise delay = 3, fall delay = 7,
// delay to high-impedance value = 13
bufif1 #(3, 7, 13) qDrive(qOut, q, enable);
```

Generally speaking the delay specification takes the form `#(d1, d2, d3)`

However, not all 3 delays always need to be specified. If only one delay is specified, it is used for all the transitions.

If two or three delays are specified then the interpretation is as follows :

From Value	To Value	2 Delays Specified	3 Delays Specified
0	1	d1	d1
0	x	min(d1, d2)	min(d1, d2, d3)
0	z	min(d1, d2)	d3
1	0	d2	d2
1	x	min(d1, d2)	min(d1, d2, d3)
1	z	min(d1, d2)	d3
x	0	d2	d2
x	1	d1	d1
x	z	min(d1, d2)	d3
z	0	d2	d2
z	1	d1	d1
z	x	min(d1, d2)	min(d1, d2, d3)

2.34 Minimum, Typical and Maximum Delays

Each of the parameters d1, d2, d3 can be specified in the form d1min:d1typ:d1max if a specification of the minimum, typical and maximum delays (due to process variations) is desired to be modeled. At simulation time, through a simulator directive, the desired delay type (minimum/typical/maximum) can be selected.

```
parameter
  R_Min = 3, R_Typ = 4, R_Max = 5,
  F_Min = 3, F_Typ = 5, F_Max = 7,
  Z_Min = 12, Z_Typ = 15, Z_Max = 17;

bufif1 #(R_Min:R_Typ:R_Max, F_Min:F_Typ:F_Max, Z_Min:Z_Typ:Z_Max)
  U1(bus, out, dir);
```

2.35 Delay Paths Across a Module

It is often useful to specify delays of paths across modules *i.e.* from pin-to-pin, apart from any gate level or other internal delays specified inside the module. The specify block allows for timing specifications to be made between a module's inputs and outputs.

```
module dEdgeFF(clock, d, clear, preset, q);

  input clock, d, clear, preset;
  output q;

// specify parameters
  specify

    specparam tRiseClkQ = 100,
      tFallClkQ = 120,
      tRiseCtlQ = 50,
      tFallCtlQ = 60;
    // module path declarations
    (clock => q) = (tRiseClkQ, tFallClkQ);
    (clear, preset *> q) = (tRiseCtlQ, tFallCtlQ);
  endspecify

// description of module's internals.

. . .

endmodule
```

Two methods are used for describing module paths.

One using => establishes a parallel connection between source's input bits (bit of a vector input or inout port) and destination's output bits (bits of a vector output or inout port). The inputs and outputs must have the same number of bits. Each bit in the source connects to its corresponding bit in the destination.

The *> establishes a full connection between source inputs and destination outputs. The source and destination need not have the same number of bits.

```
(a, b *> c, d) = 10;
```

This statement is equivalent to assuming a, b, c, and d are single-bit entities,

```
(a => c) = 10;
(a => d) = 10;
(b => c) = 10;
(b => d) = 10;
```

If e and f are both 2-bit entities, then

```
(e => f) = 10;
```

is equivalent to

```
(e[1] => f[1]) = 10;
(e[0] => f[0]) = 10;
```

2.36 Named Events

Verilog permits defining named events that allow procedural code to cause an abstract event to occur which can be sensed by other modules but needs no nets and port connections to propagate from the source module (module which has caused the named event to occur) to other modules.

```
module Eventsource();

// a named event declaration
event ready;

. . . // other declarations

always
  begin
    #100 number = number + 1 -> ready;
  end

endmodule
```

```
module topNE();

. . . // declarations

  Eventsource U1();
  Eventsense U2(a, b);

endmodule

module Eventsense(X, Y);

. . . // declarations

always
  begin
    @(U1.ready)
    h = h + 1;
  end

endmodule
```

2.37 Hierarchical Names

The scope of registers and wire declarations, and module instance names is the module in which they are defined. The hierarchical naming mechanism is used to gain access to names defined in other modules by providing their full path names.

2.38 Sequential and Parallel Blocks

The `begin ... end` block has sequential execution semantics.

The `fork ... join` block has parallel execution semantics (*e.g.* all the statements of the `fork ... join` block start execution in parallel at the current simulation time when execution reaches the `fork ... join` block). The block execution completes (and execution passes to the next sequential statement) when the execution of the most delayed instruction is complete.

2.39 System Tasks and Functions

`Verilog` provides a number of built-in system tasks and functions for the purposes of display of simulation results (as simulation proceeds), file I/O, reporting/providing current value of simulation time, generation of random numbers, stopping of simulation *etc.* The identifiers for system tasks have $ as their first character. The tasks and functions are :

```
$display, $monitor, $finish, $random, $stime, $stop,
$strobe, $write, $time, $fopen, $fclose, $fdisplay,
$fmonitor, $fstrobe, $fwrite
```

2.40 Compiler Directives, Macros, Include Files

```
`default_nettype typeOfNet
`timescale 1ns/100ps
`define SUBTRACT 6'b010101
```

Then a usage such as

```
if(aluctrl == `SUBTRACT)
```

is equivalent to

```
if(aluctrl == 6'b010101)
```

To include another `Verilog` file into current file :

```
`include "filename.V"
```

Chapter 3

Experiments Related to Language Semantics

This example shows the result of logical and case equality operators. In logical equality, result is one bit output *i.e.* '0' or '1' depending upon the condition if it is true or false. Output will be unknown if any bit is 'x' or 'z'. In case equality, there is bitwise comparision including 'x' and 'z' values. The result is true or false only.

```
module test(out1);

  output[7:0] out1;
  reg [7:0] out1;
  reg [5:0] in1;
  reg [5:0] in2;
  reg [2:0] in3;
  reg [3:0] v3,v4;
  reg v1, v2;
  reg ready;

initial
  begin
    #5 ready = 1;
    #5 in1 = 6'b101111;
    in2 = 6'bx01111;
    in3 = 2;
// #5 in1 = 8; in2 = -8; in3 = 2;
    v1 = 0; v2 =0; v3 = 4'b0000; v4 = 4'b0000;
  end

always @(in1 | in2)
  begin
    if (in1 == in2) out1 = in3;
    v1 = (in1 == in2);
    v3 = (in1 === in2);
    v2 = (in1 == in3);
    v4 = (in1 === in3);
  end
endmodule
```

This example is to test divide-by-zero problems. In case of real numbers, rounding is always done. Divide by zero of an integer and stored in an integer gives 'x'. Divide by zero of an integer and then stored in reg gives 'x'. Division by zero of a real number which then is stored in an integer gives -1. Division by zero of a real number and then stored in reg gives 1. Divide by zero of a real number and then stored in real gives inf.

For relational (logical) operations, if the logical expression is invalid then result is 'x' and condition becomes false.

```
module short_cir;

//reg a,b,c,d,e;
  integer a,b,c,f,h,d;
  reg ready,out1,d1,d2;
//reg r1[31:0],r2[31:0];
  time e;
  real p1,p2,p3;

initial
  begin
    a = 10;
    b = 10;
    c = 8;
    d = 0;
    e = 20;
    h = 2.5/0;
    d1 = 2.5/0;
    f = c/d;
    d2 = a/0;
    p1 = -5.0/0;
    p2 = 5.0/0;
    p3 = p1/p2;
//  r1 = f;
//  r2 = c/d;
    ready = 0;
    out1 = 0;
    #10 ready = 1;
  end

always @ (ready)
  begin
```

```
//   if ( (a == b) || (e >(p1)) )
     if ( 0 < p1)
//     out1 <= #5 (a/b);
       out1 = (a==b);
   end
endmodule
```

3.1 Time Control : Delay Operator, Event Control

Verilog provides 3 different types of time control constructs that can be used in a behavioural description.

3.2 Delay construct (#)

- To delay the execution of a statement.
- To delay the assignment of a computed expression value to a register.

3.2.1 # Time Control Examples

```
initial
begin
  #4 a = 1;
  #3 b = 2;
end

initial sysclock = 0;

always  #50
  sysclock = ~sysclock;
```

There is one delta delay in the assignment of wires from registers. This is true for multiple assignments from the the same input wire.

```
module siggen4(s4,s6);

  output s4,s6;
  wire s4,s6;
  wire sy,s5;

  buf u2(s5,sy);
  buf u3(s6,s5);
// not u2(s5,sy);
// not u3(s6,s5);
  siggen1 u1(sy);
  assign s4 = sy;

endmodule

module siggen1(s1);
  output s1;
  reg s1;

initial
  begin
    s1<= #10 1'b0;
    s1<= #15 1'b1;
    s1<= #20 1'b0;
    s1<= #25 1'b1;
    s1<= #30 1'b0;
  end
endmodule
```

The following code is to check the delay between nets and user defined gates. There is always one delta delay associated with the nets as opposed to the predefined Verilog gates (only one delta delay is associated with full combinational logic).

```
module inv_chain(q5);

  output q5;
  wire q5;
  wire q1, q2, q3, q4;
  reg in;

  inv U1(q1,in), U2(q2,q1), U3(q3,q2), U4(q4,q3), U5(q5,q4);

initial
  begin
    in = 0;
    #10 in = 1;
    #10 in = 0;
    #10 in = 1;
    #10 in = 0;
  end

endmodule

module inv(zout,zin);
```

```
  output zout;
  reg zout;
  input zin;

always @(zin)
  zout <= ~zin;
endmodule
```

3.3 Event Construct (@)

To make the execution of a statement wait until the specified change(s) (event) has occurred in some other part of the circuit (outside the concurrent process containing the time control statement).

3.3.1 @ Time Control Example

Commonly used forms are :

```
   @(expression)
   @(expression1 or expression2 or ...)
   @(posedge onebit)
   @(negedge onebit)
   @(event)
```

```
@(a | b) c = a ^ b; // This needs to be checked.
@(c) z = y;
```

An alternate equivalent to @(c) will be

```
@(posedge c or negedge c) a = b;

always @(a or b)
  c = a ^ b;

always @(posedge sysclock)
  Q = d;

always @(posedge sysclock)
  #5 Q = d;
```

3.4 Conditional Delay Construct (wait (condition))

To make the execution of a statement wait until the condition becomes true. If the condition is already true when the execution reaches the `wait` statement, execution continues beyond the `wait` statement and the process is not suspended. Wait is a level sensitive stand-alone statement, and is used for synchronizing two processes through handshake.

3.4.1 Wait Time Control Example

```
wait(ready);
```

3.5 Assignment Types : Procedural, Blocking, Nonblocking, Continuous

3.6 Assignments with Time Control

The # and @ time control discussed earlier precedes a statement. These forms of time control delay execution of the following statement until the specified simulation time.

There are two special kinds of assignment statements that have time control inside the assignment statement. These two forms are known as the blocking and nonblocking procedural assignments.

3.6.1 Blocking Procedural Assignment

```
var = #delay expression;
var = @(posedge onebit) expression;
var = @(negedge onebit) expression;
```

What distinguishes this from a normal instantaneous assignment is that the expression is evaluated at the current simulation time (`$time`) when the statement is reached, but the variable does not change until after the specified delay. Also (as the name implies), the execution of process is blocked (suspended) for the specified delay.

```
initial
  begin
    . . .
    a = @(posedge sysclk) b;
    . . .
  end
```

is same as

```
initial
  begin
    . . .
    temp = b;
    @(posedge sysclk) a = temp;
    . . .
  end
```

3.6.2 Nonblocking Procedural Assignment

The syntax for a nonblocking procedural assignment is identical to a blocking procedural assignments except that the assignment operator is <= instead of =.

```
var <= #delay expression;
var <= @(posedge onebit) expression;
var <= @(negedge onebit) expression;
```

For a nonblocking procedural assignment, the expression is evaluated at the current simulation time (`$time`) when the execution reaches the statement, but the variable is updated after the specified delay. However, execution continues with the next sequential statement at the current simulation time (`$time`).

```
// execute at time 8, schedule update to 1 at 16 (8 + 8).
initial #8 a <= #8 1;

// execute at time 12, schedule update to 0 at 16.(12 + 4)
initial #12 a <= #4 0;
```

3.6.3 Examples for Nonblocking Procedural Assignment

```
module test8(s);
  output s;
  reg s;

initial
  begin
    s <= #10 1'b0;
    s <= #15 1'b1;
    s <= #20 1'b0;
    s <= #25 1'b1;
    s <= #30 1'b0;
    s <= #35 1'b0;
  end
endmodule

module siggen4(s4);

  output s4;
  wire s4;

  siggen2 u2(s4);

endmodule

module siggen2(s2);

  output s2;
  wire s2;
  wire sy;

  buf u2(s2,sy);
  siggen1 u1(sy);

endmodule

module siggen1(s1);
  output s1;
  reg s1;

initial
  begin
    s1 <= #10 1'b0;
    s1 <= #10 1'b1;
    s1 <= #10 1'b1;
//  s1 <= #10 1'b0;
    s1 <= #25 1'b1;
    s1 <= #25 1'b0;
    s1 <= #25 1'b1;
    s1 <= #30 1'b1;
    s1 <= #15 1'b0;
    s1 <= #20 1'b1;
  end
endmodule
```

The example given below is used for checking the circular shift. Since `s0` and `s1` are defined as registers (and not wires) there is no delta delay involved. Delta delay is always there when assignment takes place from registers to wires.

```
module circ_shift_beh;
```

```
  reg s0, s1, clock;

initial
  begin
    s0 = 0;
    s1 = 1;
    clock = 0;
    # 50 clock =1;
    # 50 clock =0;
    # 50 clock =1;
    # 50 clock =0;
  end

always @(posedge clock)
  begin

// The two lines below work as sequential statement.
// It does not work as shift register.
//      s0 = s1;
//      s1 = s0;
// The two lines below work as nonblocking statement.
// It works as shift register.

    s0 <= s1;
    s1 <= s0;
  end
endmodule
```

3.6.4 Example : Continuous Assignment to Nets

```
assign #10 sum = aIn ^ bIn^ cIn,
  cout = (aIn & bIn) | (bIn & cIn)
       | (aIn & cIn);
```

versus

```
assign (strong0, strong1) #10
  sum = aIn ^ bIn ^ cIn,
  cout = (aIn & bIn) | (bIn & cIn)
       | (aIn & cIn);
```

3.6.5 Example : Net and Continuous Assignment Declarations

```
module modXor(AXorB, a, b);
```

```
  output AXorB;
  input a, b;
  wire #10 AXorB = a ^ b;

endmodule
```

We could also have only declared the wire in a net declaration, and then placed a continuous assignment on it through a separate assign statement as follows :

```
wire #10 AXorB;

assign AXorB = a ^ b ;

module wand_of_assigns(a, b, c);

  input a, b;
  output c;
  wand #2 c;

assign #5 c = ~a;
assign #3 c = ~b;

endmodule
```

When a delay is given in a net declaration, the delay is added to any driver that drives the net. Thus, net delay and gate delay (specified in a gate instantiation) or continuous assignments delay add-up to give the total delay between a change of input to the gate assign statement and propagation of the resulting output on the net (to the inputs of other gates).

3.7 Delay Through Combinational Logic and Nets

3.7.1 Example : Transfer of Data From Input to Output Register

Note that the input register and output register are of different sizes. If there is x or z value in the input data, then the output register has the x or z at the same bit positions. The extra bits in output register are filled by '0' only.

```
module transfer(out1);

   output[4:0] out1;
   reg [6:0] out1;
   reg [5:0] in1;
   reg ready;

initial
  begin
```

```
    #5 ready = 0;
    #6 in1 = 6'b010011; ready = 1;
    #6 in1 = 6'b100011; ready = 0;
    #6 in1 = 6'bz10011; ready = 1;
    #6 in1 = 6'bx00011; ready = 0;
  end

always
  begin
    @(in1 & ready)
//  @(in1 | ready)
     out1 = in1;
   $monitor($time,"out1=%b:in1=%b", out1, in1);
  end

endmodule
```

3.8 Example : Concatenation Operation

The data is given in integer format and the result is stored in bits. In the register, the number is treated as unsigned number for the operations.

```
module test3(out1);

  output[15:0] out1;
  reg [15:0] out1;
  reg [7:0] in1;
  reg [4:0] in2;
  reg ready;

initial
  begin
   #5 ready = 1;
   #6 in1 = 8; in2 = -8;
  end

always
  begin
    @(in1 | in2)
     out1 = {in1,in2};  -- concatenation
//   out1 = in1 + in2;
     $monitor($time,"out1=%b:in1=%b:in2=%b", out1, in1, in2);
  end

endmodule
```

3.9 Suggested Exercises

1. Contrast the traditional ASIC design flow with a HDL based design flow for a FPGA implementation.

2. List the various hardware design styles and design abstractions that Verilog HDL supports. Depict these design abstractions on Gajski's Y-chart.

3. Give five examples of hardware designs where HDL based design approach is not suitable. Explain your answer in each case.

4. How would you generate a bus, `BusMy[0:2]`, from the scalar variables, `i`, `j` and `k` ?

5. How would you form a new bus, `BusNew[7:0]`, from two other buses, `BusA[0:3]` and `BusB[7:4]` ?

6. How would you store a 16-bit wire with `1001100110011001` using replication and concatenation of `1001` ?

Chapter 4

Behavioural Modeling

Verilog describes the behaviour of a piece of hardware using one or more *concurrent processes* each containing a procedural code (a sequence of procedural statements) that describes when and how the outputs are affected as a result of change of inputs.

4.1 Behavioural Modeling Constructs

- initial statement (a one-shot process for initialization).
- always statement (an always running process).

```
initial <statement>;
always <statement>;
```

<statement> may be a single procedural statement terminated by a semicolon (;) or a compound statement defined by enclosing a sequence of procedural statements between the Verilog keywords begin and end.

```
initial
  q = 0;

always
  @(negedge clock) q = data;
```

4.2 Register Data Object for Behavioural Description

A Verilog register is analogous to a variable in procedural programming languages such as C. Registers are used by procedural statements in Verilog to describe the behaviour of hardware. However, they do not necessarily imply/specify the occurrence of corresponding hardware registers in the actual logic circuit implementation. Thus, Verilog registers, in reality, are variables used for the procedural description of the circuit behaviour.

A register may have a bit-width of one or several bits. Either a single-bit, or several contiguous bits of a vector register can be addressed and used in an expression. Each bit of a register may take the value 0, 1, x or z. Where,

- 0 – Logic Low.
- 1 – Logic High.
- x – Unknown Logic State.
- z – Tristate.

```
// 1-bit (scalar) register tempBit.
reg tempBit;

// 16-bit (vector) register tempWord.
// MSB is tempWord[15].
reg[15:0] tempWord;

// 16-bit (vector) register tempRevWord.
// MSB is tempRevWord[0].
reg[0:15] tempRevWord;
```

Either a single-bit, or several contiguous bits of a vector register can be addressed and used in an expression.

```
// Bit 5 of tempWord.
tempWord[5];

// Bit 12 through 7 of tempWord.
tempWord[12:7];
```

4.3 Example : Behavioural Description of 8-bit Parity Generator

```
module paritygen(in,z);

input[0:7] in;
output z;
reg z;
reg i;
```

```
@(in)
z = 0;
for(i = 8; i; i-1)
  z = z ^ in[i-1];

endmodule
```

4.4 Example : Behavioural Description of Simple Traffic Light Controller

```
module trafficLight;
  reg clock, red, yellow, green;
  parameter
    on = 1, off = 0,
    red_time = 300, yellow_time = 30, green_time = 200;

// initialize colours

  initial red = off;
  initial yellow = off;
  initial green = off;

  always begin
    red = on;
    light(red, red_time);
    green = on;
    light(green, green_time);
    yellow = on;
    light(yellow, yellow_time);
  end

task light;
  output colour;
  input [31:0] tick;

  begin
    repeat (tick) @(posedge clock);
    colour = off;
  end
endtask

  always begin
    #100 clock = 0;
    #100 clock = 1;
    end
endmodule
```

4.5 Suggested Exercises

1. A 3-to-8 decoder is to be used for accessing the bit that has to be changed in a 8-bit register. The bit selected by the 3-bit input is made '1' and all the remaining bits of the register are made '0'. Assume that the LSB (bit 0) is the right-most bit and MSB (bit 7) is the left-most bit of the register. Write the `Verilog` code for this.

2. A 3-to-8 decoder is used to select the bit that is to be changed in a 8-bit register. The register's bit selected by the 3-bit input is `XOR`ed with '1' and all the remaining bits of the register are `AND`ed with '1's. Assume that the LSB (bit 0) is the right-most bit and MSB (bit 7) is the left-most bit of the 8-bit register. Write a `Verilog` code for this circuit.

3. Write a behavioral `Verilog` code for a circuit that counts the number of 0s and 1s in a 8-bit input pattern. If the number of 1s and 0s are odd, a flag (called `odd_flag`) is set, cleared otherwise. If the number of 1s and 0s are even, a flag (called `even_flag`) is set, cleared otherwise. If the number of 1s and 0s are of different parity, a flag (called `parity_flag`) is set, cleared otherwise. If the number of 1s is greater than number of 0s, a flag (called `one_flag`) is set, cleared otherwise. If the number of 0s is greater than the number of 1s, a flag (called `zero_flag`) is set, cleared otherwise. If the number of 1s and 0s is equal, both the `one_flag` and `zero_flag` are set.

4. Show a behavioral model of a synchronous circuit that counts the number of 1s in a continuous stream of bit data that consists of 1s and 0s which arrive at each positive clock edge. When the count of 1s reaches 128, it outputs a signal that turns on a red light for a period equal to five numbers of positive clock edges. The red light is then switched-off and the default state of green light on is set.

5. A 3-out-of-5 code detector is to be designed. The device receives a 5-bit parallel input word. The detector output is a logic 1 for any code word that has exactly three zeros and is logic 0 otherwise. Declare a `Verilog` module for the detector and write its algorithmic behavioral code.

6. Give the behavioral model for an ALU that operates on integer values. The operations that this ALU supports are ADD (addition), SUB (subtraction), MUL (multiplication), DIV (division), LT (less than), LE (less than or equal to) and EQ (equal to) on these integers. The output of the ALU for arithmetic operations is ALU_OUT and for logical operations is ALU_LOG.

7. Develop the behavioural model for a 8-input multiplexer with ports of 1-bit width and an inertial delay from data or select input to data output of 5 ns.

8. A coin-operated ticketing machine accepts coins of one-rupee, two-rupee and five-rupee through its coin slot. A detector tells the machine whether the coin inserted is a one-rupee coin, a two-rupee coin or a five-rupee coin.

As soon as the value of the coins inserted is exactly Rs. 3, the machine gives out a ticket and is ready to serve the next customer. If the coins inserted add up to more than the Rs. 3, the machine gives out a ticket and returns the right change by releasing appropriate coins and is ready to serve another customer.

Write a behavioral `Verilog` model for the above ticketing machine.

9. Write the function for converting integer values to an 8-bit vector.

10. Write a task for reversing the contents of a byte *i.e.* bit7 is swapped with bit0, bit6 is swapped with bit1, and so on.

11. Write a behavioral `Verilog` code for a circuit that counts the number of 1s in a 8-bit input pattern. If the number of 1s is less than 4, then a flag (called `under_flag`) is set, cleared otherwise. If the number of 1's is more than 4, then a flag (called `over_flag`) is set, cleared otherwise. If the number of 1s is equal to 4, then both the flags, (`under_flag` and `over_flag`), are set.

12. Write a behavioral `Verilog` code for a digtal clock with the following features :

    - It should have the ability to display 12-hour as well as 24-hour time.
    - The 12-hour mode should show a.m. or p.m. appropriately.
    - The seconds display should be optional.
    - The time is updated on the clock display every second if the seconds display option is enabled, otherwise the clock display is updated every minute.
    - The user should be able to set the clock to a desired time.
    - The clock starts as soon as it is powered-on in the 24-hour mode with 00:00:00 (note that seconds display is enabled) as the time.

Chapter 5

Data-flow Modeling

Different ways of modeling an 8-bit parity generator (without a register to store the parity value) :

5.1 Example : Data-flow Description Using a Single Continuous Assignment Statement

```
module paritygen(in,z);

input[0:7] in;
output z;

assign #2
z = in[0] ^ in[1] ^ in[2] ^ in[3] ^ in[4] ^ in[5] ^ in[6] ^ in[7];

endmodule
```

5.2 Example : Data-flow Description Using Multiple Continuous Assignment Statements

```
module paritygen_4(in,z);

input[0:7] in;
output z;

wire w1, w2;

assign#1
```

```
w1 = in[0] ^ in[1] ^ in[2] ^ in[3] ^ in[4],
w2 = in[5] ^ in[6] ^ in[7],
z = w1 ^ w2;

endmodule
```

5.3 Example : D Flip-flop

```
module dEdgeFF(q, clock, data);

output q;
reg q;
input clock, data;
wire clock, data;

initial
  q = 0;

always
  @(negedge clock) #10 q = data;

endmodule
```

Flip-flop based on NAND gates.

```
module ffNand;

wire q, qBar;
reg preset, clear;

nand #1 g1(q, qBar, preset), g2(qBar, q, clear);

endmodule
```

5.4 Suggested Exercises

1. Develop a data-flow model of an 8-bit odd parity checker. The parity checker has eight inputs, i1 to i8, and an output p. The logic equation of the parity checker is:

$$p = ((i_1 \oplus i_2) \oplus (i_3 \oplus i_4)) \oplus ((i_5 \oplus i_6) \oplus (i_7 \oplus i_8))$$

   Code the above logic equation by using single statement, two statements, three statements, *etc.*

Chapter 6

Structural Modeling

Verilog describes the constructional details of a module in terms of its constituent modules.

The basic constructs for structural description in Verilog are :

- Instances of user defined modules.
- Instances of pre-defined gate level primitives. Such instances can also specify drive strength and gate delay.
- Nets (of various types) to interconnect module instance ports and gate instance ports.

6.1 Examples

This example shows the instantiation of a module in top module. Order of names in instantiation port list must match the order of ports in the definition.

```
module Top(q_top,qbar_top);

  output q_top,qbar_top;
  reg R,S;

  rs_latch U1(R,S,q_top,qbar_top);

initial
  begin
    R = 1; S = 0;
    #10
    R = 0; S = 1;
    #10
    R = 0; S = 0;
    #10
    R = 1'bx; S = 0;
  end
endmodule

module rs_latch(r,s,q,qbar);

  input r,s;
  output q, qbar;
  reg  q, qbar;

always @(r or s)
  begin

    if((r==1) & (s==0))
    begin
      q = 0;
      qbar = 1;
    end
    else if((r==0) & (s==1))
    begin
      q = 1;
      qbar = 0;
    end
    else if ((r==0) & (s==0))
      ;
    else
    begin
      q = 1'bx;
      qbar = 1'bx;
    end
  end
endmodule
```

This example shows an adder that is made up of a half-adder and a full-adder.

```
module adder (sum, a, b);

  input a, b;
  output sum;
  wire[1:0] a, b;
  wire[2:0] sum;
  wire c;

  half_adder ha1(c, sum[0], a[0], b[0]);
  full_adder fa1(sum[2], sum[1],
                 a[1], b[1], c);

endmodule
```

6.2 Example : Structural Description Using Instantiation of a Single 8-input XOR Gate

```
module paritygen_1(in, z);

input[0:7] in;
output z;

xor #2 U1(z, in[0], in[1], in[2], in[3], in[4], in[5], in[6], in[7]);

endmodule
```

6.3 Example : Structural Description Using Instances of 4-input and 2-input XOR Gates

```
module paritygen_2(in, z);

input[0:7] in;
output z;

wire w1, w2;

xor #1
  U1(w1, in[0], in[1], in[2], in[3]),
  U2(w2, in[4], in[5], in[6],in[7]),
  U3(z, w1, w2);

endmodule
```

One can similarly write other structural descriptions using XOR gates with different number of inputs.

6.4 Suggested Exercises

1. How can you use an instance of a 4-input multiplexer as a 2-input multiplexer ? Please explain your answer.

2. Show a 4-input muliplexer that is made up of two numbers of 2-input multiplexers and an OR gate.

3. Develop a complete structural model of a 4-input multiplexer that is made up of gates. These could be any combination of AND, NAND, OR, NOR, NOT gates.

4. Develop a structural model of an 8-bit odd parity checker using instances of an XOR gate. The parity checker has eight inputs, `i1` to `i8`, and an output `p`. The logic equation of the parity checker is:

$$p = ((i1 \oplus i2) \oplus (i3 \oplus i4)) \oplus ((i5 \oplus i6) \oplus (i7 \oplus i8))$$

Chapter 7

RT-Level Modeling

In the hierarchy of design abstractions, RTL represents that level of design detail where the designer has already decided upon the "detailed architecture" of his design.

By "detailed architecture" we mean that the designer has already fixed the following :

- All the registers he is going to use in his design.
- All the functional blocks he is going to use in his design
- How under the controls issued by a Finite State Machine (FSM) data stored in specific registers will be supplied to functional blocks to form Boolean expressions that will then be stored in specified registers on a clock-cycle by clock-cycle basis.

In fact, the transfer of data from one or more source registers to a destination register after processing by a functional unit that lies in the transfer path is called a *register transfer* step.

A RTL design description, therefore, captures all the architectural blocks of the design *e.g.* registers, functional units (ALUs, Barrel shifters, Multipliers, *etc.*) and multiplexers and/or buses interconnecting them. It enumerates all the interface points of each architectural block (data inputs, data outputs, control inputs, and clock – where applicable) and describes the functioning of the blocks behaviorally. Also included in the design is the control generation block – the FSM that generates controls for all the blocks in the architecture.

7.1 Architecture Blocks

Blocks in the architecture can be categorized in the following four categories:

1. Fixed function combinational function blocks needing no controls (*e.g.* adders, multipliers).
2. Multi-function combinational function blocks needing controls to select the function to be performed (*e.g.* ALUs).
3. Clocked (synchronous) storage elements needing no controls (D-Flip-Flops based registers, latch based storage elements).
4. Clocked storage elements needing control to activate their synchronous and/or asynchronous behaviors (D Flip-Flop based registers with enable and synchronous / asynchronous Clear and/or Preset).

The FSM generating controls itself can be described using three architectural blocks:

1. Clocked storage elements (the state register) usually with asynchronous reset.
2. The combinational function block for generation of next state.
3. The combinational function block for generation of controls (outputs).

The functionality of items at (2) and (3) above can also be combined in a single combinational function block.

Additional clocked storage elements can be used to store the external inputs and/or control outputs of the FSM necessitated by timing and glitch prevention requirements on inputs and/or control outputs.

7.2 Suggested Exercises

1. What are the characteristics of a RTL model ?
2. For specifying the two inputs to a pipelined 32-bit multiplier, a 4-to-16 decoder is used twice to select one out of the sixteen 16-bit registers for specifying the memory addresses which contain the multiplicand and the multiplier. Write a Verilog code fragment to achieve this.

Chapter 8

Mixed-Style Modeling

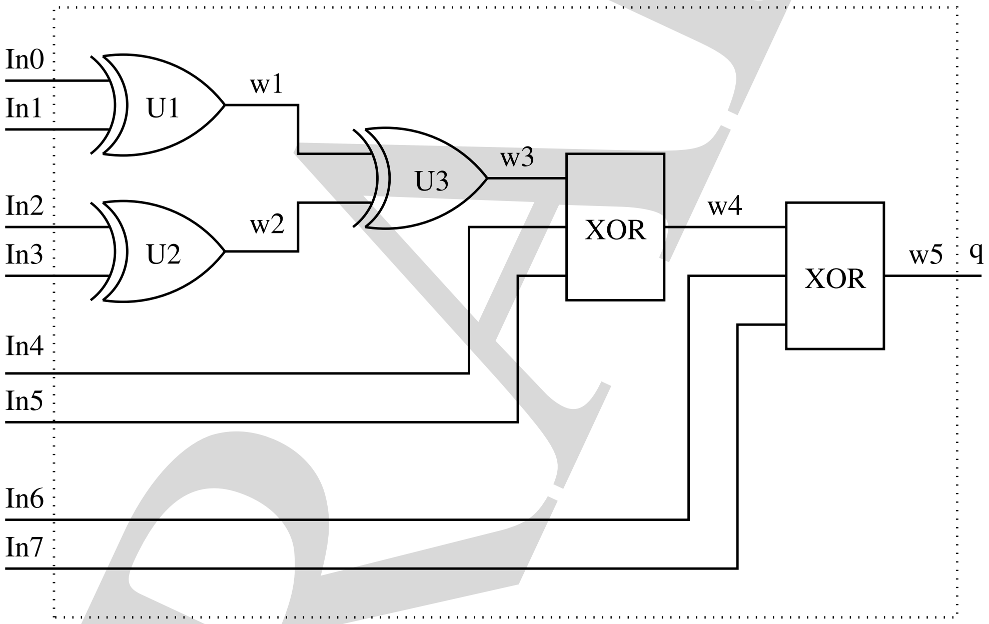

Figure 8.1: **8-bit Parity Generator**

8.1 Example : 8-bit Parity Generator

This example models a 8-bit parity generation logic whose output is registered at the rising edge of the clock. It also illustrates a mixing of all the three basic modeling styles in a single module.

```
module parity_register(in, clock, q);

input[0:7] in;
input clock;
```

```
output q;
reg q;

wire w1, w2, w3, w4, w5;

xor #1
  U1(w1, in[0], in[1]),
  U2(w2, in[2], in[3]),
  U3(w3, w1, w2);

assign #2
  w4 = w3 ^ in[4] ^ in[5],
  w5 = w4 ^ in[6] ^ in[7];

initial
  q = 0;

always @(posedge clock)
  q = w5;

endmodule
```

In this code the XOR gate with a gate delay of one time unit has been instantiated three times. `U1`, `U2`, `U3` are the names of the three instances. In the `U1` instance the output of the XOR gate has been connected to wire `w1` (declared in the module) and the two inputs of the gate have been connected to module inputs `in[0]` and `in[1]`.

Similarly, one can look at the connections of instances `U2` and `U3` of the XOR gate and realize that these three instantiations are the structural description of a network of three XOR gates arranged in two levels of logic as shown below :

The set of two continuous assignment statements with a delay of two time units constitutes the data-flow description of two Boolean expressions that respectively drive wires, `w4` and `w5`, with their values. Whenever the logic value of any wire, input port or register in the right-hand side expression of a continuous assignment changes, the expression evaluates its value and drives the wire on the left-hand side of the assignment with the evaluated value of the expression.

The order of continuous assignments in a module is unimportant — it does not decide the sequential order in which the assignments will be executed. True to the spirit of data-flow modeling where the central concept is that a change of data value of a data item (such as a `wire`, a `register` or a `port`) automatically flows to and forces the evaluation of all the expressions that contain that data item; the execution order of continuous assignment statements is determined by the order of occurrence of changes in the values of the registers, wires or ports in their right-hand side expressions. Several continuous assignment statements execute concurrently if their right-hand side expressions contain the same data item, or if the different data items in their respective right-hand side expressions change at the same time.

The `initial` statement and its associated block of sequential statements (which in the example comprises of only one sequential statement, `q = 0;`) are the constructs for a behavioural descriptions that is executed only once — at the start of the simulation. They are used to initialize the system state at the beginning of the simulation run. In the above example, the `initial` statement initializes register `q` to logic value `0` at the start of the simulation.

The `always` statement along with an event sensitive construct (*e.g.* `@posedge(clock)`) and the associated block of sequential statements (here only one statement, `q = w5;`) are the constructs for behavioural description that is always executed whenever a positive edge of the clock occurs. In the above example, at every positive edge of the clock, register `q` is assigned the logic value of wire `w5`.

The order of execution of statements in a block of sequential statements associated with an initial or an always statement is decided by the order in which those statements appear inside the block.

The statement outside the sequential blocks are all concurrent statements. Thus, in the above example, the three XOR gate instantiation statements, the two continuous assignment statements, the initial statement and the always statement are all concurrent statements. Only the initial and always concurrent statements are allowed to have an associated block of sequential statements.

For simulation, each concurrent statement creates its own process — with its own separate control thread and synchronization points that synchronize further execution of the process with the events generated by other processes. Thus, the construct `@(posedge clock)` creates a synchronization point. Execution of the process stays halted at this point until such time when a positive edge appears on the clock signal as a result of change in the logic value of the clock signal. This change in value of the clock signal must be caused by an external agent — that is from outside of this process, such as a user interactively changing the value of clock signal or another process changing its value.

All module ports are nets of type `wire`. A behavioural model having only `initial` and `always` statements whose associated blocks of sequential statements can assign values only to registers and cannot assign values to output ports (because ports are wires) drive values onto output ports of modules via declaring a register of the same name as the name of the output port of the module, and assigning values to this register which implicitly drives the port.

While the above example shows the flexibility of `Verilog` in permitting mixing of description styles in a single module, usually modules are written using a single description style.

Chapter 9

Coding of State Machines and Sequential Logic

9.1 Example : NAND-based Latch

The NAND-based latch has two inputs, `preset`, `clear`, and two outputs, `q` and `qbar`. This example uses the structural modeling style for the latch description.

```
module ffNand;

  wire q, qbar;
  reg preset, clear;

nand

g1(q, qbar, preset),
g2(qbar, q, clear);

initial
  begin
    #5
    $monitor($time,,"preset=%b:clear=%b:q=%b:qbar=%b\n",
             preset, clear, q, qbar);
    preset = 1;
    clear = 0;
  end
endmodule
```

9.2 Example : D Flip-flop

```
module dEdgeFF(clock, d, clear, preset, q);

  input clock, d, clear, preset;
  output q;

  specify

    specparam
      tRiseClkQ = 100,
      tFallClkQ = 120,
      tRiseCtlQ = 50,
      tFallCtlQ = 60;
    (clock => q) = (tRiseClkQ, tFallClkQ);
    (clear, preset *> q) = (tRiseCtlQ, tFallCtlQ);

  endspecify

// description of module's internals.

endmodule
```

9.3 Example : A Three Counter

```
module three_count(out, in, clock, reset);

  output out;
  input in, clock, reset;
  reg out;
  reg[1:0] currentState, nextState;

always@(posedge clock or negedge reset)
begin // the sequential part
  if(~reset)
    currentState <= 0;
  else
    currentState <= nextState;
end

always@(in or currentState)
begin // the combinational part
  out = 0;
  nextState = 0;
  if(currentState == 0)
    if(in) nextState = 1;
  if(currentState == 1)
    if(in) nextState = 2;
  if(currentState == 2)
    begin
      if(in) nextState = 0;
      else nextState = 1;
    end
end

endmodule
```

9.4 Suggested Exercises

1. Give a binary to Gray code converter using FSM abstraction. Code this as a Verilog program.
2. Show the GCD (Greatest Common Divisor) computation algorithm using FSM abstraction. Code this as a Verilog program.
3. Do addition of two 8-bit integers by repeatedly using a 1-bit full adder.
4. Implement a 8-bit multiplier which makes use of a 2-bit full adder (shift-add alogorithm).
5. Model a traffic light controller.
6. Model a microwave oven controller.
7. Model a washing machine controller.
8. Model a elevator controller.
9. Model a vending machine for coffee/tea/cold-drink/tickets/...
10. A coin-operated ticketing machine accepts coins of fifty-paisa, one-rupee and two-rupee through its coin slot. A detector tells the machine whether the coin inserted is a fifty-paisa coin, a one-rupee coin or a two-rupee coin.

    As soon as the value of the coins inserted is exactly two rupees and fifty paise, the machine gives out a ticket and is ready to serve the next customer. If the coins inserted add up to more than the ticket charge, the machine gives out a ticket and returns the right change by releasing appropriate coins and is ready to serve another customer.

    Write a behavioral Verilog model for this FSM (of Mealy type) controller.
11. Model an ATM using FSM abstraction.
12. Model a BCD up/down counter using FSM abstraction.
13. Model a binary up/down counter using FSM abstraction.

14. Model a digital 24-hour clock with alarm(s) using FSM abstraction.

15. Write a `Verilog` code for an alarm clock with the following features :

    - The underlying clock is a 24-hour clock with hours, minutes and seconds.
    - The clock starts as soon as it is powered-on in the 24-hour clock mode with 00:00 (hours:minutes) as the time.
    - The user should be able to set the current time (only hours and minutes) of the 24-hour clock. This is the local-time mode.
    - The user is be able to set the alarm to a desired time. The alarm will beep continuously for 30 seconds when the alarm time is reached. The user can shut the alarm by switching-off the alarm mode.
    - The snooze mode is available only when the alarm is beeping. It shuts-off the beeping alarm immediately. It then gives the beeping alarm after a lapse 10 minutes. The snooze mode can be used only once by the user after the alarm starts to beep.
    - The stop-clock mode allows the user to count the elapsed time since the time the stop-clock was enabled. The display in this mode shows hours, minutes as well as seconds in 00:00:00 form. This is the only mode of the clock in which seconds are displayed.
    - The other-time mode allows the clock to be set for another time zone.
    - The current time is user-selectable to be the local-time or the other-time. The alarm operates on the current time that is selected by the user.

16. Show a FSM model of a synchronous circuit that counts the number of 1s in a continuous stream of bit data that consists of 1s and 0s which arrive at each positive clock edge. When the count of 1s reaches 128, it outputs a signal that turns on a red light for a period equal to five numbers of positive clock edges. The red light is then switched-off and the default state of green light on is set.

Chapter 10

Coding of Test Benches

To test the Verilog model of any digital hardware, we need to apply test stimuli waveforms to its inputs, capture the response waveforms at the model's outputs and compare the captured response waveforms with the expected output waveforms.

A Verilog model written for the above purpose essentially simulates the running of a test on a hardware test setup with the Device Under Test (DUT) plugged in the DUT socket — and is typically called a *test bench*.

One can use Verilog not only for modeling the device (DUT) but also for modeling the Stimulus Generator (usually as behavioural Verilog code) as well as Response Capture and Response Analysis Block (also as behavioural Verilog code) *i.e.* the entire test bench (see Figure 10.1).

Example : A typical test bench template.

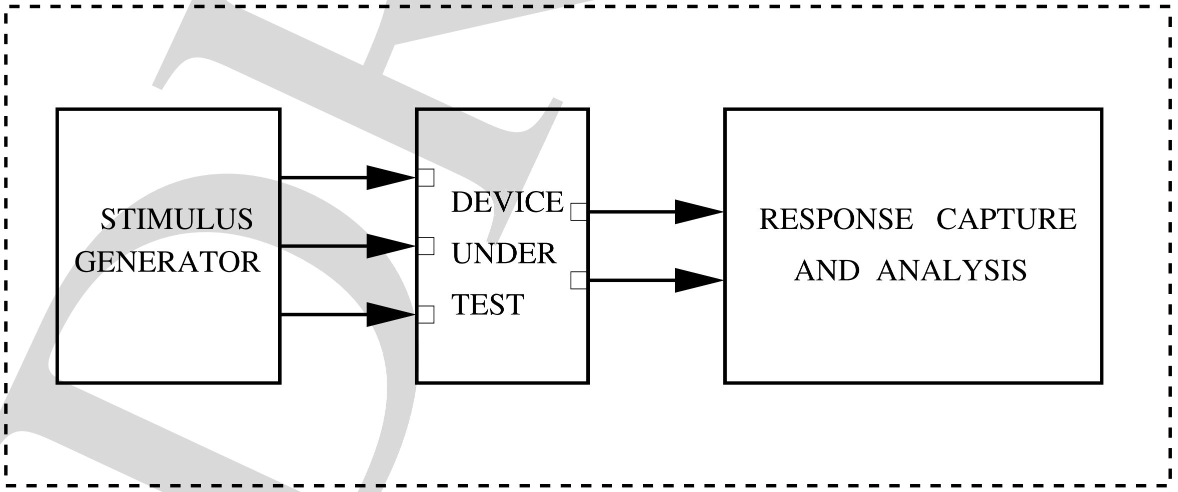

Figure 10.1: **Test Bench Blocks**

```
module testBench();

// initial process for initialization, if required.

// always process to generate the stimulus
// waveforms on signals that will connect to the DUT's
// inputs (Verilog behavioural model).

// Declare local signals that will connect
// to the DUT's I/O's.
// call the DUT such that these signals connects
// to the DUT's inputs.

// always process to monitor and compare the
// captured response waveforms (Verilog behavioural model)
// from the DUT's outputs by having all the local signals
// that connect to the DUT's outputs.

endmodule
```

10.1 Suggested Exercises

1. Write test benches for the various models written earlier.

Chapter 11

Gate Level Modeling

A gate level model of a circuit describes the circuit in terms of interconnections of logic primitives *e.g.* `AND, OR, XOR`. This allows the designer to describe the actual implementation in terms of elements found in a technology library or data handbook.

11.1 Example : A 1-bit Full Adder

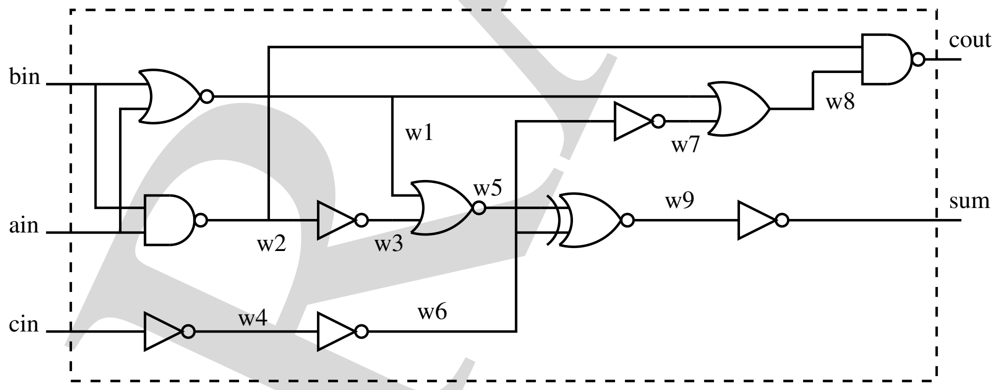

Figure 11.1: **1-bit Full Adder**

```
module fullAdder(cout, sum, ain, bin, cin);

output cout, sum;
input ain, bin, cin;

wire w2; // What about other internal wires ?

  nand (w2, ain, bin), (cout, w2, w8);
  xnor (w9, w5, w6);
  nor (w5, w1, w3), (w1, ain, bin);
  or (w8, w1, w7);
  not (sum, w9), (w3, w2), (w6, w4), (w4, cin), (w7, w6);

endmodule
```

Chapter 12

Switch Level Modeling

The switch level of modeling provides a level of abstraction between the logic (gates) and analog-transistor levels of abstraction. The switch level transistors are modeled as being either on or off (conducting or not conducting). Instead of the whole range of analog voltages (or currents), a small number of discrete values are used, called *signal strengths*.

12.1 Example : A MOS Shift Register

```
module shiftreg(out, in, phase1, phase2);

output out;
input in, phase1, phase2;

tri wb1, wb2, out; // tri nets pulled up to VDD
pullup (wb1), (wb2), (out); // Depletion mode pullup transistors

trireg (medium) wa1, wa2, wa3; // Charge storage nodes

supply0 gnd; // Ground supply

nmos #3 a1(wa1, in, phase1), b1(wb1, gnd, wa1),
        a2(wa2, wb1, phase2), b2(wb2, gnd, wa2),
        a3(wa3, wb2, phase1), gout(out, gnd, wa3);

endmodule
```

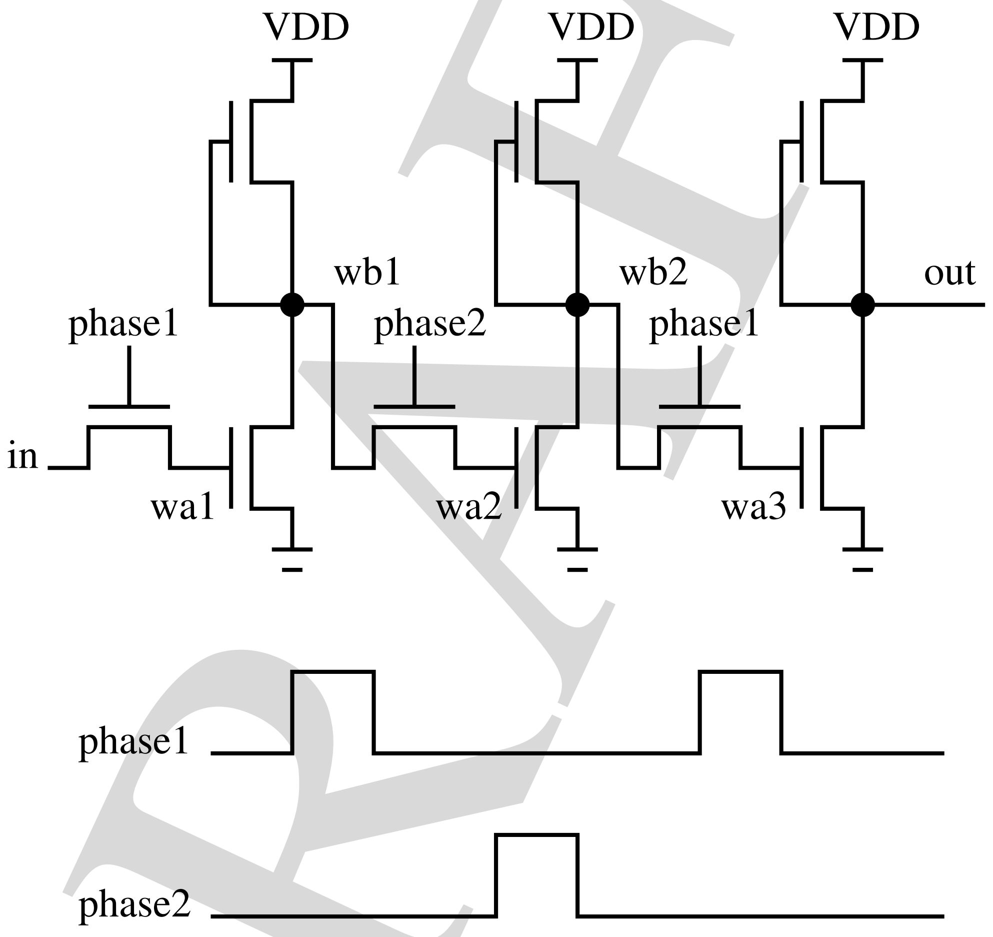

Figure 12.1: **MOS Shift Register**

Chapter 13

Coding Style for Synthesis

`Verilog` tasks are synthesizable as long as there are no timing controls in the body of the `task`.

Most of the synthesis tools interpret `'x'` as “don’t care.”

13.1 Writing Synthesisable RTL Code

While writing synthesisable RTL code, care must be taken to use only those constructs that are synthesisable.

13.1.1 # Delay Statements

The `# delay` construct is ignored by synthesis tools.

13.1.2 Combinational Function Blocks

In particular, if the process includes `if` or `case` statements (which make assignments selectively) one must take special care to ensure that an activation of the process results in an assignment to every single output and not just a selected few of them. If some outputs are not assigned values in certain activations of the process, latches would be inferred for them to maintain their old values. Such unintended latches must be avoided through a careful examination of the `if` and `case` statements used in the program code.

13.1.3 Storage Elements

Remember! The order in which controlling conditions occur in an `if` statement decides the priority of execution of their corresponding controlled statements – as at most one group of controlled statements corresponding to the highest priority controlling statement that is TRUE are executed when an `if` statement executes.

Thus, by placing the condition(s) of activation of asynchronous behavior ahead of the condition of activation of synchronous behavior in the `if` statement, overriding priority of an asynchronous control inputs is maintained for both simulation and synthesis.

In the sequential `if` statement, the condition of activation of asynchronous behavior should be included first and only then should the condition of activation of synchronous behavior follow.

Chapter 14

Entering Design Constraints and Synthesis

14.1 RTL and Logic Synthesis

Currently, RTL designs are created and validated by designers. A logic synthesis tool (such as `Design Compiler` or `FPGA Compiler` from Synopsys or `BuildGates` from Cadence) is then used to automatically generate the gate level design and output the net-list for a specific technology library.

Besides the RTL design, a priority of design objectives (whether to design for maximum speed or minimum area), the level of effort permitted for design optimization (low, medium, high) and a set of constraints are also input to the logic synthesizer.

The constraints may take the simple form of specification of an overall clocking rate for the design or the designer may constrain delays along specified paths running across one or more blocks of the RTL design. Additional constraints like maximum permissible fan-ins and fan-outs for gates to be used, maximum permissible rise and fall times for nodes in the circuit, preclusion of / preference for certain cells in the target technology library *etc.* can also be provided to the logic synthesis tool.

While constraints can be supplied interactively to the synthesis tool, usually they are included in a separate constraint file which is read in by the synthesis tool or they are part of a synthesis script.

The synthesis tool then translates the RTL design into a gate level net-list while trying to satisfy all the constraints supplied by the designer. Detailed timing reports on the generated net-lists are provided by the synthesis tool along-with a list of constraints that could not be satisfied (if that is the case).

If some constraints are not met, then the designer should either relax the constraints, or try 'high' effort of optimization, or alter his architecture and rewrite the RTL code or part thereof.

The synthesis tools typically work in two phases:

1. In the first phase, a design using generic gates is synthesized.
2. In the second or *technology mapping* phase, the generic gates are replaced by cells in the specified technology library.

After the placement-and-routing of the generated net-list, delays from the layout are extracted and back-annotated into the net-list. Timing checks are once again carried out on the back annotated net-list to see if the timing constraints are still satisfied post-layout.

If some timing constraints are not satisfied post-layout, then either a place-and-route iteration or a synthesis iteration followed by a fresh place-and-route are carried out until timing closure is achieved.

In the deep sub-micron and nanometric technologies, wire delays have become far more important than the gate delays. Consequently achieving timing closure becomes a problem. As a result, a new generation of synthesis tools has been developed that integrates the logic synthesis and physical synthesis (place-and-route) into a single tool to achieve timing closure faster.

14.2 Tool-Specific Tasks

Explore your design tool and check how to generate timing reports, getting CLB/Gate usage reports, and selecting suitable FPGA device (Xilinix) for design implementation.

Chapter 15

Mini-project Ideas

15.1 Suggested Topics

Please note that many exercises suggested under Section 9.4 can also be adapted as mini-projects.

1. Behavioral and RTL models of i8085 processor.
2. Behavioral and RTL models of i8086 processor.
3. Behavioral and RTL models of MC6800 processor.
4. Behavioral and RTL models of MC68000 processor.
5. Behavioral and RTL models of DLX processor.
6. Behavioral and RTL models of various RISC processors.
7. Behavioral and RTL models of a bus arbitrator.
8. Behavioral and RTL models of a cache memory controller.
9. Behavioral and RTL models of a DRAM memory controller.
10. Behavioral and RTL models of simple electronic telephone exchange.
11. Behavioral and RTL models of a Fibonacci number generator.
12. Behavioral and RTL models of specialized functions processor.
13. Behavioral and RTL models of IEEE-754 compliant floating-point arithmetic processor.
14. Behavioral and RTL models of various communications controllers.

15. Behavioral and RTL models of various peripheral controllers as per Intel Data Handbook.

16. Behavioral and RTL models of various peripheral controllers as per Motorola Data Handbook.

17. Write a `Verilog` code for a digital radio-cum-alarm appliance with the following features :

    - The radio-part is a FM radio and it can store 5 radio stations frequencies. It can tune any FM station between 88 MHz to 108 MHz in steps of 0.05 MHz.
    - The user is be able to set the alarm to a desired time. The alarm will beep continuously for 1 minute when the alarm time is reached. However, if the radio option is selected for alarm, then the appliance plays the radio station that is chosen by the user while setting the alarm.
    - The snooze mode is available only when the alarm is beeping. It shuts-off the beeping alarm immediately. It then gives the beeping alarm after a lapse 15 minutes.
    - The sleep mode is available only when the radio is playing. It shuts down the radio after the user's choice of 10, 20 or 30 minutes has elapsed.
    - The user should be able to set the current time (hours and minutes) of the 24-hour clock.
    - The underlying clock is a 24-hour clock with hours and minutes.
    - The appliance starts as soon as it is powered-on in the 24-hour clock mode with 00:00 as the time.

18. The following is the specification given for an alarm clock chip :

    - The underlying clock is a 24-hour clock with hours, minutes and seconds.
    - It has a calendar of 100 years (starting from the year 2000). In the calendar mode, it displays the current day, month and year. In the set calendar mode, the user can set the date.
    - The clock starts as soon as it is powered-on in the 24-hour clock mode with 00:00 (hours:minutes) as the time and January 1, 2000 as the date.
    - The user should be able to set the current time (only hours and minutes) of the 24-hour clock. This is the local-time mode.
    - The user is be able to set the alarm to a desired time. The alarm will beep continuously for 20 seconds when the alarm time is reached. The user can shut the alarm by switching-off the alarm mode.
    - The snooze mode is available only when the alarm is beeping. It shuts-off the beeping alarm immediately. It then gives the beeping alarm after a lapse 10 minutes. The snooze mode can be used only once by the user after the alarm starts to beep.

- The stop-clock mode allows the user to count the elapsed time since the time the stop-clock was enabled. The display in this mode shows hours, minutes as well as seconds in 00:00:00 form. This is the only mode of the clock in which seconds are displayed.
- The other-time mode allows the clock to be set for another time zone.
- The current time is user-selectable to be the local-time or the other-time. The alarm operates on the current time that is selected by the user.
- It is also possible to set the local-time mode to 12-hour mode in which AM or PM is displayed appropriately.

Show the methodology and design steps that you will follow with appropriate `Verilog` code to design such an alarm clock chip.

Chapter 16

`Verilog`-Related Books

1. M. G. Arnold, `Verilog` *Digital Computer Design : Algorithms to Hardware*, Prentice-Hall, 1999.
2. L. Bening and H. Foster, *Principles of Verifiable RTL Design*, Kluwer, 2001.
3. J. Bergeron, *Writing Testbenches*, Second Edition, Kluwer, 2003.
4. J. Bhasker, *A* `Verilog` *HDL Primer*, Second Edition, Star Galaxy, 1999. (Cheap Edition)
5. J. Bhasker, `Verilog` *HDL Synthesis : A Practical Primer*, Star Galaxy, 1998. (Cheap Edition)
6. M. D. Ciletti, *Modeling, Synthesis and Rapid Prototyping with* `Verilog` *HDL*, Prentice-Hall, 1999.
7. K. Coffman, *Real World* `Verilog` *FPGA Design*, Prentice-Hall, 1999.
8. U. Golze, *VLSI Chip Design with the HDL* `Verilog`, Springer-Verlag, 1996.
9. IEEE Standard 1364-2001, `Verilog` *Langauge Reference Manual*, IEEE Press, 2001.
10. IEEE Standard 1364-1995, `Verilog` *Langauge Reference Manual*, IEEE Press, 1996.
11. J. M. Lee, `Verilog` *Quick Start*, Kluwer, 2002.
12. W. F. Lee, `Verilog` *Coding for Logic Synthesis*, Wiley, 2003.
13. Z. Navabi, `Verilog` *Digital System Design*, McGraw-Hill, 1999.
14. T. R. Padmanabhan and B. B. T. Sundari, *Design Through* `Verilog` *HDL*, Wiley, 2003.
15. S. Palnitkar, `Verilog` *HDL : A Guide to Digital Design and Synthesis*, Second Edition, Pearson/Prentice-Hall, 2003.

16. S. Palnitkar, Verilog *HDL : A Guide to Digital Design and Synthesis*, Prentice-Hall, 1996. (Cheap Edition)

17. V. Sagdeo, *The Complete* Verilog *Book*, Kluwer, 1998.

18. D. J. Smith, *HDL Chip Design : A Practical Guide*, Doone Publisher, 1997.

19. D. R. Smith and P. D. Franzon, Verilog *Styles for Synthesis of Digital Systems*, Pearson/AW, 2001.

20. E. Sternheim. R. Singh and Y. Trivedi, *Digital Design with* Verilog *HDL*, Automata Publisher, 1990.

21. S. Sutherland, *Verilog-2001 : A Guide to New Features*, Kluwer, 2002.

22. D. E. Thomas and P. R. Moorby, *The* Verilog *Hardware Description Language*, Fifth Edition, Kluwer, 2002.

23. R. M. Zeidman, Verilog *Designer's Library*, Prentice-Hall, 1999.

Appendix A

Verilog-2001 *versus* Verilog-1995

Please check your simulation/synthesis tool's compatibility and default settings related to Verilog-1995 and Verilog-2001. The major differences to note between the two are :

1. Verilog-1995 required all 1-bit nets, driven by a continuous assignment, that are not declared to be ports, to be declared explicitly. It is the only 1-bit net type that must be declared. This anamoly has been done away with in Verilog-2001.

```
module andor(y, a, b, c);
  output y;
  input a, b, c;
  wire n; // this is not required in Verilog-2001

  assign n = a & b;
  assign y = n | c;
endmodule
```

2. In Verilog-1995, or is used as a separator in the sensitivity list (note that you cannot use and in the sensitivity list). Verilog-2001 allows , to be used as a separator to better indicate the sensitivity list elements.

3. In Verilog-1995, ports of a module were needed to be declared many times. Verilog-2001 makes this declaration closer to the ANSI C kind of declaration. The example given below illustrates this and the sensitivity list change mentioned above.

Verilog-1995

```
module dFFarl(q, d, clk, rst_l);
  output q;
  input d, clk, rst_l;
  reg q;

  always @(posedge clk or negedge rst_l)
```

```
    if(!rst_l) q = 1'b0;
    else q = d;
endmodule
```

Verilog-2001

```
module dFFarl(
  output reg q,
  input d, clk, rst_l);

  always @(posedge clk, negedge rst_l)
    if(!rst_l) q = 1'b0;
    else q = d;
endmodule
```

4. The word, register, used in Verilog-1995 is meant to describe the group of variable data types — reg, integer, real, time and realtime. To remove any misunderstanding that may arise due to the use of the word "register" in the context of hardware design, which means something different, Verilog-2001 uses the word "variable" wherever "register" is used in Verilog-1995.

5. Verilog-2001 has the ** operator.

6. Verilog-2001 allows multi-dimensional arrays.

7. Verilog-2001 has a generate statement.

8. Verilog-2001 has standardized the configuration statements. They now offer better control of binding files to instances in a design. New keywords and constructs like library, design, config, endconfig, default, liblist and instance have been added.

9. Verilog-2001 allows signed arithmetic.

10. Verilog-2001 has enhanced file input/output capabilities.

11. Verilog-2001 allows re-entrant task for recursion.

12. Verilog-2001 has generate statements.

13. Verilog-2001 introduced the @* operator for the sensitivity list to model/synthesisize combinational logic. Many designers found that they were getting extra storage elements instead of "pure" combinational logic due to having missed out a particular input or forgetting to take care of all the possible values for a particular signal. The synthesis tools build combinational logic strictly from the expressions inside of an always block. The always @* usage eliminates the need to list every single input in the sensitivity list. Nets and variables that appear on the right-hand side of assignments, in function/task calls, or case/if expressions are included in this implicit sensitivity list construct. Even though @(*) is allowed, it is not recommended for use as many vendors don't support this to avoid confusion with (*...*) statement structure.

```
module decoder(
  output reg[7:0] y,
  input [2:0] a,
  input en);

  always @* begin
    y = 8'hff;
    y[a] = !en;
  end
endmodule
```

Appendix B

More About Verilog-2001

Verilog-2001 is the latest version of the IEEE 1364 Verilog HDL and PLI standard. The IEEE 1364-2001 standard contains two major parts :

1. The definition of a Verilog HDL.
2. The definition of a Verilog Programming Language Interface (PLI).

The Verilog HDL part contains the keywords, syntax and semantics to model hardware circuits. Software tools can then use these models to simulate hardware behavior, synthesize functional models into structural netlists, *etc.*.

The Verilog PLI part provides an interface so that engineers can customize and extend the capabilities of software tools such as Verilog simulators.

B.1 Resources and Web Sites

B.2 Verilog Related Organizations/Conferences

http://www.eda.org/ is the site for EDA Industry Working Groups for VHDL, Verilog *etc.*.

http://www.accellera.org is the Accellera site formed after the merger of VHDL International and Open Verilog International.

http://www.hdlcon.org/ is for information on HDL Conferences.

`http://www.codes-symposium.org/` is for CODES (Codesign) Symposium/Workshop.

`http://www.ecsi.org/` is the site of European Electronic Chips and Systems Design Initiative (ECSI).

`http://www.si2.org/ic/` is the site of Silicon Integration Initiative (Si2) Industry Council.

B.3 `Verilog` Resource Sites

`http://www.ovi.org` is for Open Verilog International.

`http://www.eda.org/vlog-synth/` is the WG on Verilog Synthesis.

`http://www.icarus.com/eda/verilog` is the site of Icarus Verilog (Free Verilog Compiler and Simulator).

B.4 `Verilog` Commercial Vendors Sites

`http://www.doulos.com/` is the Doulos HDL Resource site.

`http://www.hdlsolutions.com/` is the HDL Solutions site.

`http://www.ftlsystems.com/` is the FTL Systems site.

`http://www.sutherland-hdl.com` is Sutherland's HDL (Verilog Resources) site.

`http://www.verilog.com/` is the VerilogCom site.

`http://www.verilog-2001.com/` is the Verilog 2001 related site.

`http://www.verisity.com/` is the site of Verisity Design.

`http://www.veritools-web.com/` is the site of VeriTools.

`http://www.wellspring.com/` is the site of Well Spring.

`http://www.whdl.com/` is the site of Willamette HDL Courseware.

B.5 Procuring a Copy of the Verilog-2001 Standard

Verilog is an IEEE standard; the official Verilog-2001 standard can only be obtained through the IEEE.

B.6 Earlier Verilog Standards

1984 : Startup company Gateway Design Automation began selling a digital simulator, called Verilog-XL. There were no standard hardware description languages at that time, so Gateway created a proprietary language for their simulator, called Verilog.

1989 : Gateway Design Automation was acquired by Cadence Design Systems. By then, the Verilog-XL simulator, with its proprietary Verilog language, had become one of the most popular simulators for ASIC design. Several companies had licensed the right to use the Verilog language independent of the Verilog-XL simulator, including Synopsys, Logic Automation, and LMSI.

1990 : Cadence released the Verilog language to the public domain, to allow any software company to use the Verilog language, without requiring a special license. Cadence retained (and till recently sold), the Verilog-XL product. Open Verilog International (OVI) was formed to control and promote the use of the Verilog language. OVI labeled the first public domain Verilog HDL as Verilog HDL 1.0.

1993 : OVI released Verilog HDL 2.0, which contained several minor enhancements to the HDL, and a major change to the Verilog PLI. OVI then submitted a request to the IEEE to make Verilog an IEEE standard.

1995 : The IEEE released the IEEE 1364-1995 Verilog standard. The primary goal of the Verilog Standard Group for Verilog-1995 was to standardize the way Verilog was used at that time. No enhancements to Verilog were considered for the 1995 standard.

2001 : The IEEE ratified the IEEE 1364-2001 Verilog-2001 standard. This latest Verilog standard contains many major and minor enhancements to the Verilog HDL and the Verilog PLI.

B.7 Backward Compatiblity With Verilog-1995

In general, Verilog-2001 is backward compatible with Verilog-1995 and earlier Verilog standards. Models which were written with previous generations should run fine on simulators, synthesis compilers or other tools which support the Verilog-2001 standard. However, there are three areas which may not be fully backward compatible. These are :

1. New reserved keywords in Verilog-2001.
2. Change in how unsized numbers are extended.
3. change in the multi-channel file descriptor returned by $fopen.

Verilog-2001 adds several new keywords to the Verilog language. Any older models which might happen to use one of these new reserved words will not work with a Verilog-2001 simulator or other software tool. This was given serious consideration on the IEEE 1364 standards committee. The standards committee tried to select keywords which are intuitive, but not very likely to have been used as design signal names. It was discussed — and rejected — to provide some sort of backward compatibility mode in Verilog-2001. The committee felt that it should be left to software tools to provide a backward compatibility option, and not be a requirement of the language. It is anticipated that most software tools will provide such an option.

The second backward compatibility issue involves the automatic extension of logic Z or X values in an assignment statement. Verilog-2001's automatic Z or X extension of an unsized right-hand side expression to any vector size on the left hand side of an assignment, is not compatible with Verilog-1995, which zero extends after the first 32 bits. The IEEE 1364 standards committee felt that this "feature" in older Verilog standards was a bug that needed to be fixed. Existing Verilog models probably do not rely on this bug, because it does not behave like hardware. The work around for the bug — to explicitly specify the vector size on the right-hand side expression — is fully compatible with Verilog-2001, so existing models that are coded correctly will behave the same with Verilog-2001 software tools.

The third possible backward compatibility problem is that Verilog-2001 reduces the maximum number of files that can be represented with a "Multi-Channel Descriptor" (MCD) from 31 to 30 files. In Verilog-1995, each call to $fopen returns a 32-bit integer MCD with a single bit set. Bit 0 is reserved as the simulator output channel and simulator log file, leaving 31 additional files that can be represented. MCD's can be ORed together, making it possible to write to multiple files at the same time. Verilog-2001 "steals" bit 31, reducing the number of MCDs to 30. In Verilog-2001, bit 31 represents a "File Descriptor" (FD) which will have bit 31 and at least one other bit set. An FD is a single-channel descriptor, and cannot be ORed with other FD's or MCD's. As many as 2**30 FD files can be opened at the same time, which greatly increases the number of files that can be represented. In addition to more open files, an FD can be used for both reading and writing to the file (an MCD can only be used for writing) and an FD file can be read or written as a binary or ASCII file (MCD's only support ASCII files). Since the FD takes bit 31 of an MCD, there will only be a backward compatibility problem if an existing application opens more than 30 files. The IEEE 1364 standards group felt the trade-off of losing 1 MCD in order to a gain virtually unlimited number of open files, reading from files, and binary file support, was worth the backward compatibility trade-off.

Please see Appendix A for more details on Verilog-2001 versus Verilog-1995.

B.8 Software Tools Support for Verilog-2001

Many of the new Verilog-2001 features are supported by various vendors of simulation and synthesis products.

ModelSim version 5.5 onwards has implemented several of the new features, and future releases have added more features.

Synopsys supports many of the features in Verilog-2001 in their VCS tool set.

Cadence supports Verilog-2001 in their NC-Verilog, BuildGates and some other Verilog-related tool sets.

B.9 Creation of the Verilog-2001 Standard

A standards committee determined what enhancements went into Verilog-2001, and what enhancement requests were rejected. The standards committee, officially called the IEEE 1364 Verilog Standards Group or VSG, consisted of about 20 active members, plus several additional members who did not actively participate in meetings, but reviewed drafts of the standard and made recommendations. The VSG members comprised engineers from a wide variety of companies, including simulator companies, synthesis companies, ASIC vendors, and hardware designers. All participation on the VSG was voluntary; the members were not paid.

The VSG met 8 to 10 times each year for about 3 years to create the Verilog-2001 standard. The VSG meetings were open meetings, which could be attended by anyone, whether an active member of the VSG, or not. Any person attending a meeting could express opinions and make proposals, however to have voting rights on proposals required attending a certain percentage of meetings each year. In general, the VSG meetings would alternate between phone conferences and in-person meetings, held in San Jose, California (in-person meetings could also be attended by conference call).

The VSG was subdivided into three task forces. The Behavioral Task Force was responsible for reviewing the RTL and behavioral portions of Verilog, and proposing changes and enhancements to the VSG. The ASIC Task Force was responsible for proposing enhancements to the gate level aspects of Verilog. The PLI Task Force proposed enhancements to the Verilog PLI. These task forces would often meet two or three times a month, usually by phone conference. All proposals made by the task forces were voted on, and sometimes ammended, by the main VSG.

B.10 Checking `Verilog-2001` Compliance

The `IEEE 1364 Verilog Standards Group` considered putting together a test suite for simulation and synthesis tools, but the proposal was rejected. The official policy is that the role of the `Verilog` Standards Group is to specify the `Verilog` standard, and respond to question regarding the standard. The VSG is not a police force, however, and does not test products or judge whether a product is IEEE compliant.